# from Interactive Theorem Proving to Integrated Theorem Proving

Makarius Wenzel

November 2016

# Abstract


Interactive theorem proving was historically tied to the read-eval-print loop, with sequential and synchronous evaluation of prover commands given on the command-line. This user-interface technology was adequate when Robin Milner introduced his LCF proof assistant in the 1970s, but today it severely restricts the potential of multicore hardware and advanced IDE front-ends.

The Isabelle Prover IDE breaks this loop and retrofits the read-eval-print phases into an asynchronous model of document-oriented proof processing. Instead of feeding a sequence of commands into the prover process, the primary interface works via edits over immutable document versions. Execution is implicit and managed by the prover in a timeless and stateless manner, making adequate use of parallel hardware.

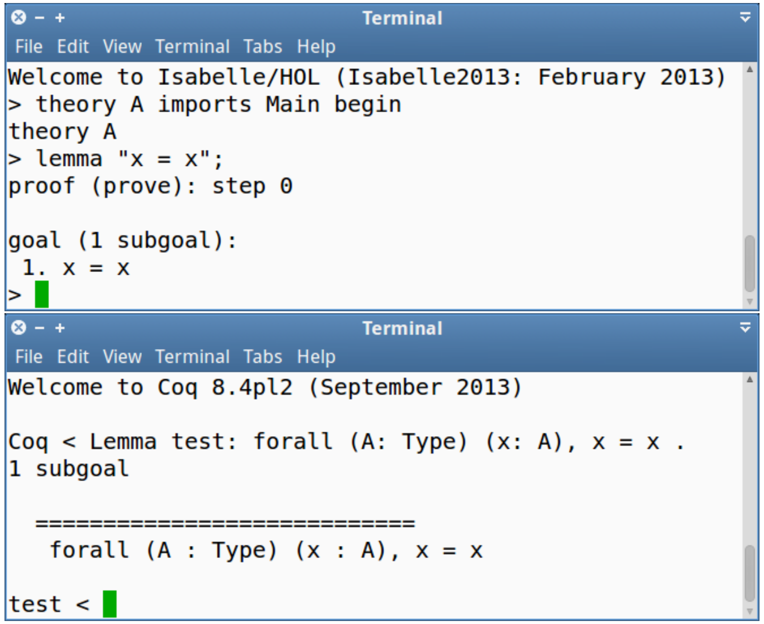
PIDE document content consists of the theory sources (with dependencies via theory imports), and auxiliary source files of arbitrary user-defined format: this allows to integrate other languages than Isabelle/Isar into the IDE. A notable application is the Isabelle/ML IDE, which can be also applied to the system itself, to support interactive bootstrapping of the Isabelle/Pure implementation.

Further tool integration works via "asynchronous print functions" that operate on already checked theory sources. Thus long-running or potentially non-terminating processes may provide spontaneous feedback while the user is editing. Applications range from traditional proof state output (which often consumes substantial run-time) to automated provers and dis-provers that report on existing proof document content (e.g. Sledgehammer, Nitpick, Quickcheck in Isabelle/HOL). It is also possible to integrate "query operations" via additional GUI panels with separate input and output (e.g. for manual Sledgehammer invocation or find-theorems).

Thus the Prover IDE orchestrates a suite of tools that help the user to write proofs. In particular, the classic distinction of ATP and ITP is overcome in this emerging paradigm of Integrated Theorem Proving.

# History

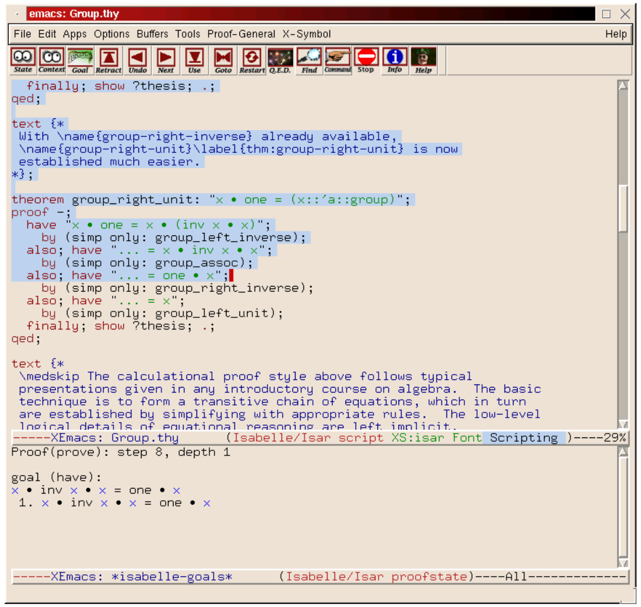
# TTY loop (≈ 1979)

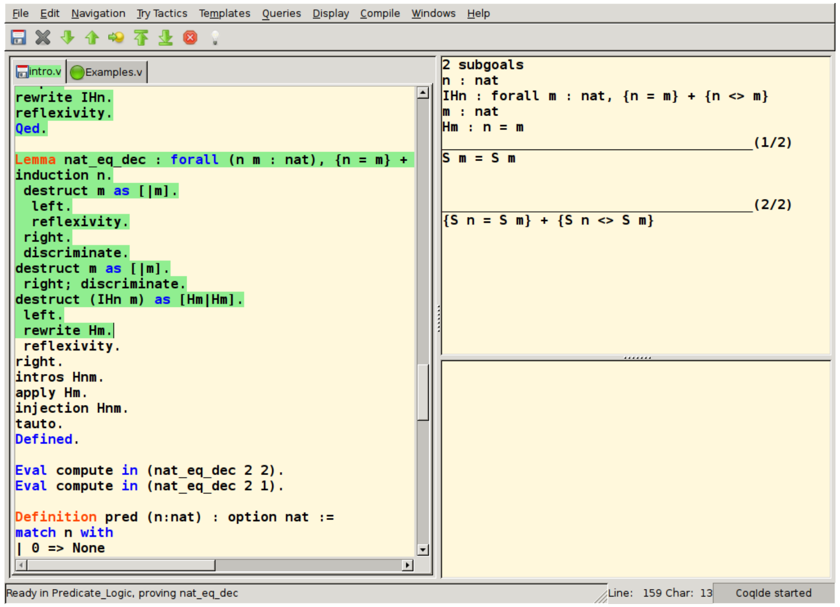



(Wikipedia: K. Thompson and D. Ritchie at PDP-11)

- user drives prover, via manual copy-paste
- inherently synchronous and sequential

# Proof General and clones (≈ 1999)




- user drives prover, via automated copy-paste and undo
- inherently synchronous and sequential

# PIDE: Prover IDE ($\approx$ 2009)

**Approach:**

**Prover** supports asynchronous document model natively

**Editor** continuously sends source edits and receives markup reports

**Tools** may participate in document processing and markup

**User** constructs document content — assisted by
GUI rendering of cumulative PIDE markup

# PIDE: Prover IDE ($\approx$ 2009)

**Approach:**

**Prover** supports asynchronous document model natively

**Editor** continuously sends source edits and receives markup reports

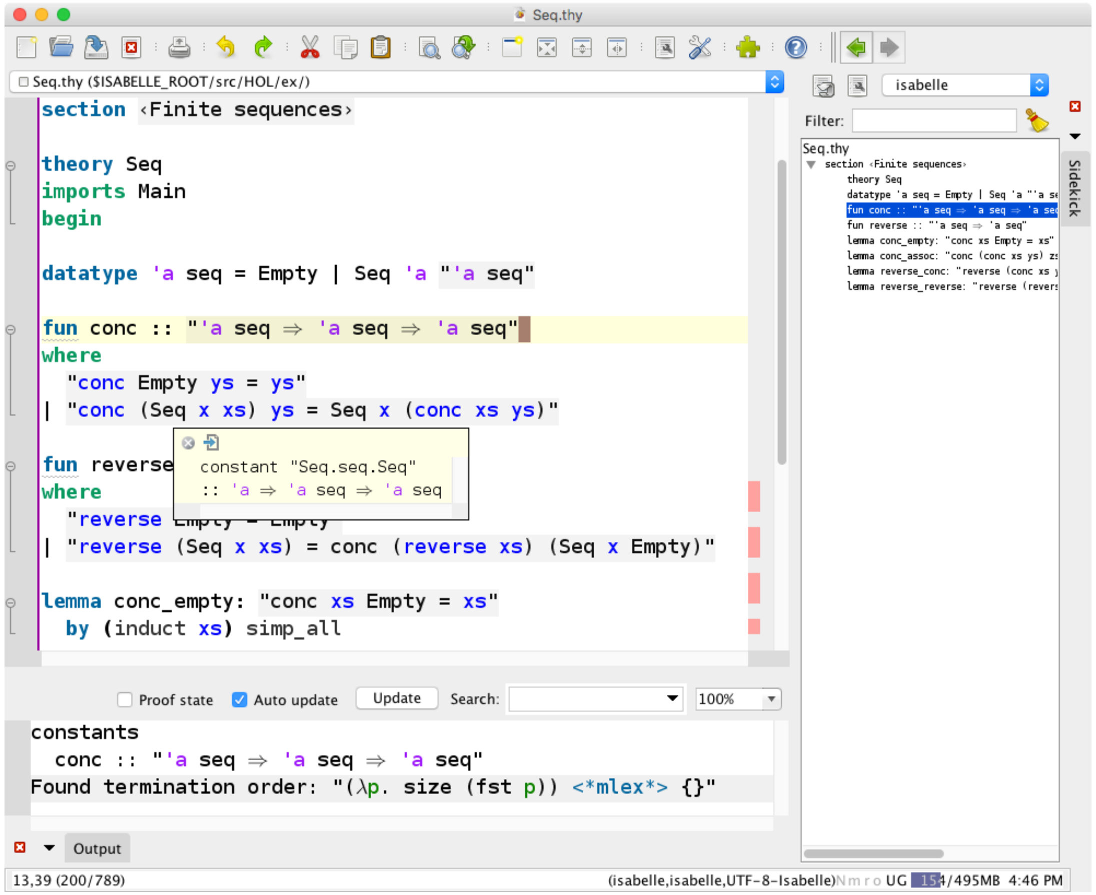
**Tools** may participate in document processing and markup

**User** constructs document content — assisted by
GUI rendering of cumulative PIDE markup

**Challenge:** introducing genuine interaction into ITP

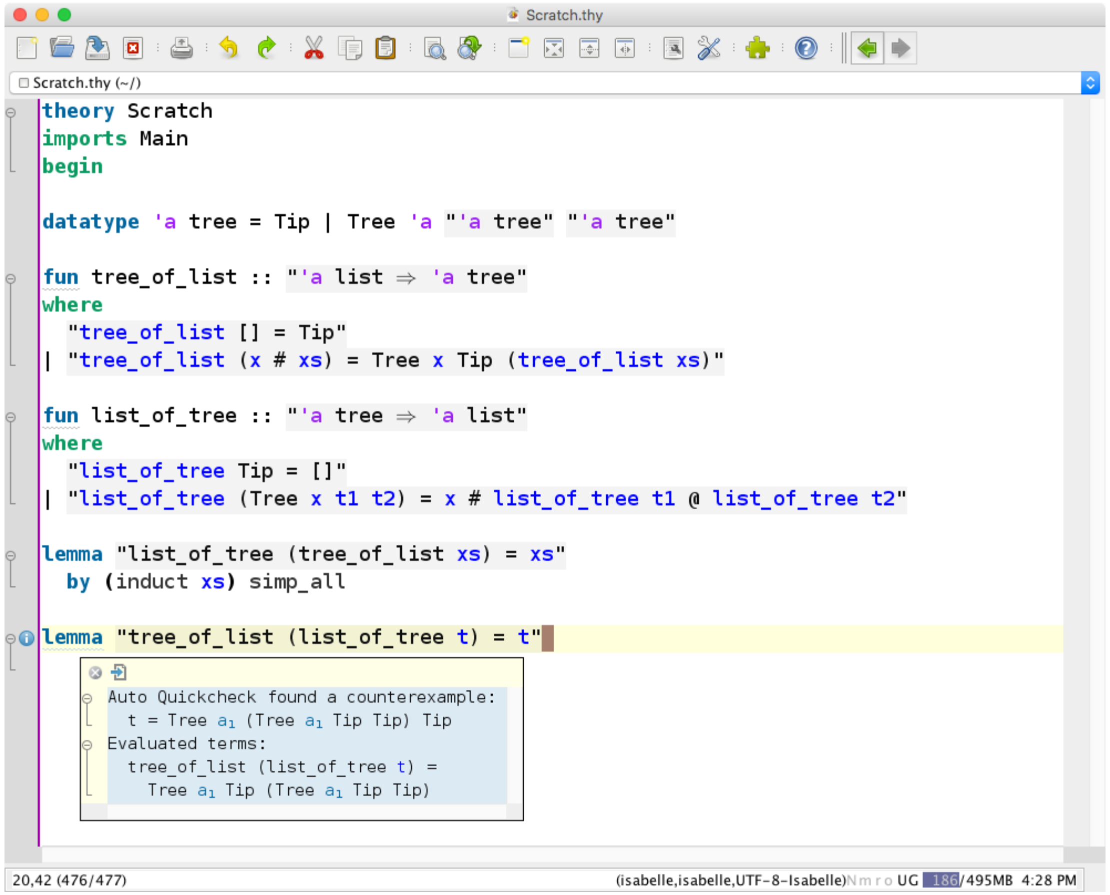
- many conceptual problems
- many technical problems
- many social problems

# Isabelle/jEdit Prover IDE (2016)




⟶ advanced user interaction

# Automatically tried tools (2016)




⟶ advanced tool integration

# Isabelle/PIDE building blocks

**jEdit:** sophisticated text editor implemented in Java
`http://www.jedit.org`

**Scala:** higher-order functional-object-oriented programming on JVM
`http://www.scala-lang.org`

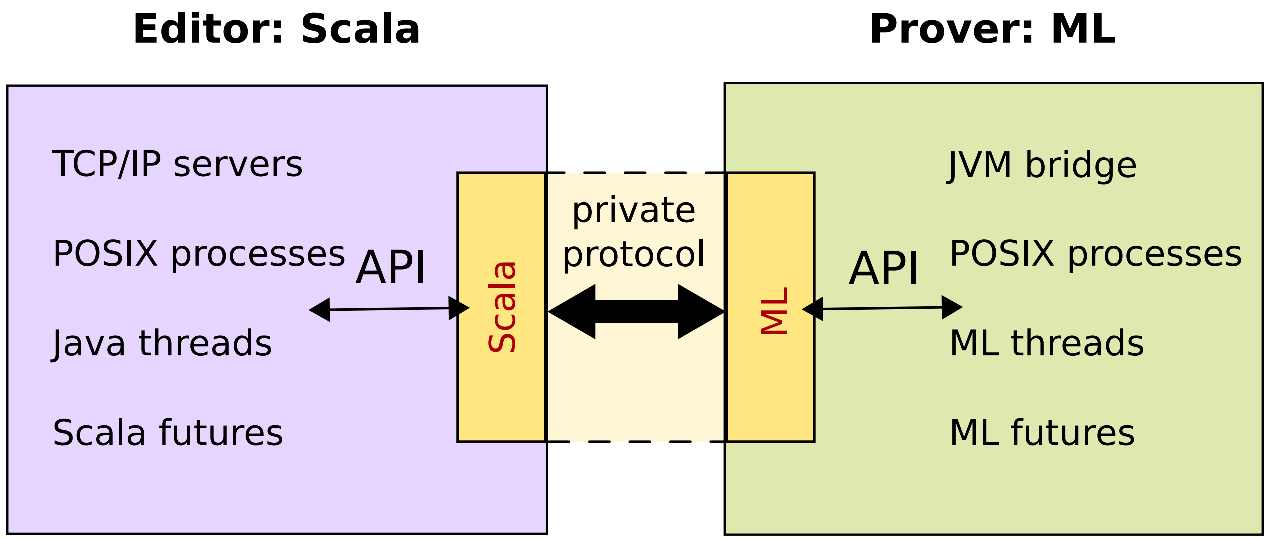
**PIDE:** general framework for Prover IDEs based on Scala
with parallel and asynchronous document processing

**Isabelle/jEdit:**

- main example application of the PIDE framework
- default user-interface for Isabelle
- filthy rich client: 2 cores + 4 GB RAM minimum

# PIDE architecture

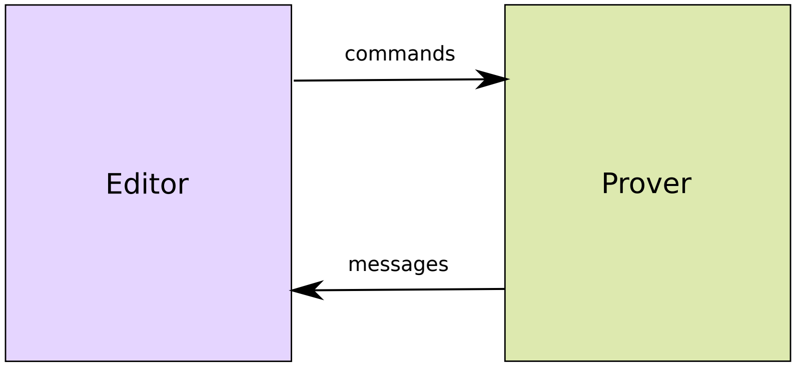
# The connectivity problem




## Design principles:

- private protocol for prover connectivity
  (asynchronous interaction, parallel evaluation)
- public Scala API
  (timeless, stateless, static typing)

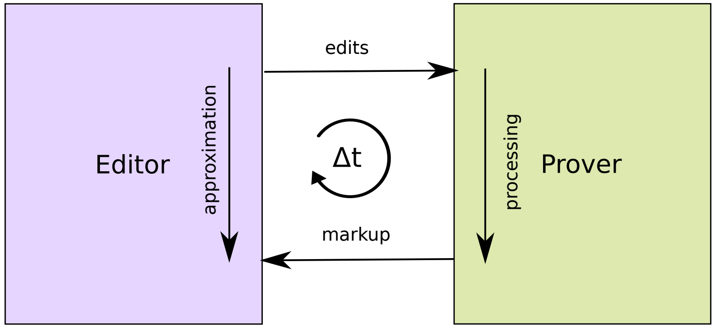
# PIDE protocol functions




- $type\ protocol\_command = name \to input \to unit$
- $type\ protocol\_message = name \to output \to unit$
- outermost state of protocol handlers on each side (pure values)
- asynchronous streaming in each direction

$\longrightarrow$ editor and prover as stream-procession functions

# Approximative rendering of document snapshots




1. editor knows text $T$, markup $M$, and edits $\Delta T$ (produced by user)
2. apply edits: $T' = T + \Delta T$ (immediately in editor)
3. formal processing of $T'$: $\Delta M$ after time $\Delta t$ (eventually in prover)
4. temporary approximation (immediately in editor):
   $\tilde{M} = \mathit{revert}\ \Delta T; \mathit{retrieve}\ M; \mathit{convert}\ \Delta T$
5. convergence after time $\Delta t$ (eventually in editor):
   $M' = M + \Delta M$

# Document content and execution

# Prover command transactions

- "small" toplevel state $st$: $Toplevel.state$
- command transaction $tr$ as partial function over $st$
  we write $st_0 \longrightarrow^{tr} st_1$ for $st_1 = tr\ st_0$
- general structure: $tr = read;\ eval;\ print$

**Interaction view:**

$tr\ st_0 =$
  **let** $eval = read\ ()$ **in** — $read$ does not require $st_0$
  **let** $st_1 = eval\ st_0$ **in** — main transition $st_0 \longrightarrow st_1$
  **let** $() = print\ st_1$ **in** $st_1$ — $print$ does not change $st_1$

**Important:** purely functional transactions with managed output

# Command scheduling

## Sequential R-E-P Loop:

$$st_0 \quad \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} \quad st_1 \quad \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} \quad st_2 \quad \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} \quad st_3 \quad \cdots$$

# Command scheduling

## Sequential R-E-P Loop:

$$st_0 \quad \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} \quad st_1 \quad \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} \quad st_2 \quad \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} \quad st_3 \quad \cdots$$

## PIDE 2011/2012:

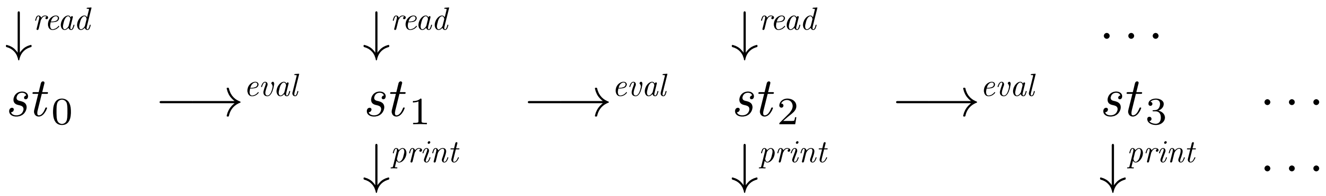
$$\begin{array}{ccccccccc}
\downarrow^{read} & & \downarrow^{read} & & \downarrow^{read} & & \cdots & \\
st_0 & \longrightarrow^{eval} & st_1 & \longrightarrow^{eval} & st_2 & \longrightarrow^{eval} & st_3 & \cdots \\
 & & \downarrow^{print} & & \downarrow^{print} & & \downarrow^{print} & \cdots
\end{array}$$

# Command scheduling

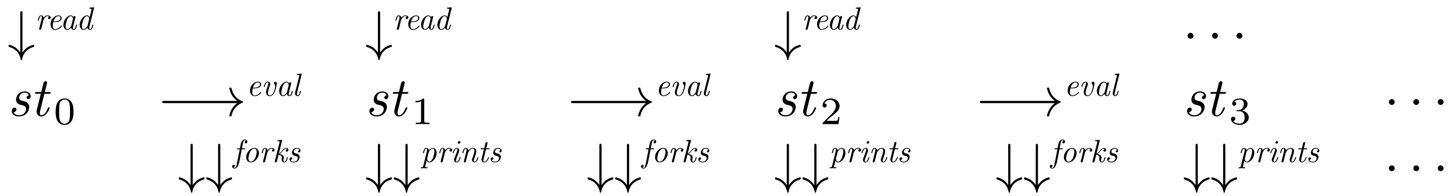
## Sequential R-E-P Loop:

$$st_0 \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} st_1 \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} st_2 \xrightarrow{read} \xrightarrow{eval} \xrightarrow{print} st_3 \cdots$$

## PIDE 2011/2012:




## PIDE 2013/2014:



# Document nodes

**Global structure:** directed acyclic graph (DAG) of theories

**Local structure:**

**entries:** linear sequence of command spans,
with static $command\_id$ and dynamic $exec\_id$

**perspective:** visible and required commands,
according to structural dependencies

**overlays:** print functions over commands (with arguments)

# Document nodes

**Global structure:** directed acyclic graph (DAG) of theories

**Local structure:**

**entries:** linear sequence of command spans,
with static $command\_id$ and dynamic $exec\_id$

**perspective:** visible and required commands,
according to structural dependencies

**overlays:** print functions over commands (with arguments)

**Notes:**

- for each document version, the command exec assignment identifies results of (single) $eval\ st$ or (multiple) $print\ st$
- the same execs may coincide for different versions
- non-visible / non-required commands remain unassigned

# Document edits

**Key operation:** *update* $\rightsquigarrow$ *assignment*

**datatype** $edit = Dependencies \mid Entries \mid Perspective \mid Overlays$
**val** $Document.update$: $version\_id \to version\_id \to$
$(node \times edit)\ list \to state \to$
$(command\_id \times exec\_id\ list)\ list \times state$

**Notes:**

- document update restructures hypothetical execution
- command exec assignment is acknowledged quickly
- actual execution is scheduled separately

$\longrightarrow$ protocol thread remains reactive with reasonable latency

# Execution management

## Prerequisites:

- native threads in Poly/ML (D. Matthews, 2006 . . . )
- future values in Isabelle/ML (M. Wenzel, 2008 . . . )

## Execution in PIDE 2013/2014:

**Hypothetical execution:** lazy execution outline with symbolic assignment of $exec\_ids$ to $eval$ and $prints$

**Execution frontiers:** conflict avoidance of consecutive versions

$Execution.start\colon unit \to execution\_id$
$Execution.discontinue\colon unit \to unit$
$Execution.running\colon execution\_id \to exec\_id \to bool$

**Execution forks:** managed future groups within execution context

$Execution.fork\colon exec\_id \to (unit \to \alpha) \to \alpha\ future$
$Execution.cancel\colon exec\_id \to unit$

# PIDE backend implementation

# PIDE protocol layers (1)

**Bidirectional byte-channel:**

- pure byte streams with block-buffering
- high throughput
- TCP socket; not stdin/stdout

**Message chunks:**

- explicit length indication
- block-oriented I/O

**Text encoding and character positions:**

- reconcile ASCII, ISO-Latin-1, UTF-8, UTF-16
- unify Unix / Windows line-endings
- occasional readjustment of positions

# PIDE protocol layers (2)

## YXML transfer syntax:

- markup trees over plain text
- simple and robust transfer syntax
- easy upgrade of text-based application

## XML/ML data representation

- canonical encoding / decoding of ML-like datatypes
- combinator library for each participating language, e.g. SML:

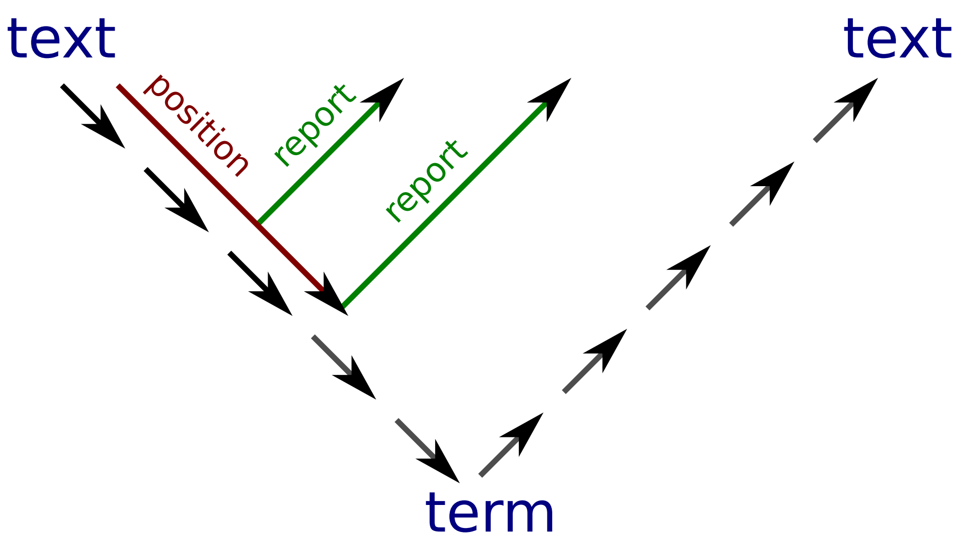
  $type\ \alpha\ Encode.T = \alpha \to XML.tree\ list$
  $Encode.string\colon string\ Encode.T$
  $Encode.pair\colon \alpha\ Encode.T \to \beta\ Encode.T \to (\alpha \times \beta)\ Encode.T$
  $Encode.list\colon \alpha\ Encode.T \to \alpha\ list\ Encode.T$

- untyped data representation of typed data
- typed conversion functions

# Markup reports

**Problem:** round-trip through several sophisticated syntax layers

**Solution:** execution trace with markup reports




**Example:** semantic markup for domain-specific formal languages

# Conclusions

# Achievements

**Renovation and reform . . .**

of Interactive/Integrated Theorem Proving
for new generations of users

**Paradigm shift . . .**

adequate use of multicore hardware with pervasive parallelism

**Document-oriented approach . . .**

for user interaction and tool integration

$\longrightarrow$ Towards *The Archive of Formal Proofs* as one big document!

## Lessons learned

- Substantial reforms of LCF-style theorem proving are possible, with big impact on infrastructure, but little impact on existing tools.
- Parallel processing is relatively easy, compared to the difficulties of asynchronous user interaction and tool integration.
- Real-world frameworks like JVM/Swing impose technical side-conditions and challenges, notably for multi-platform support.

# Try it yourself!

**Current release:** February 2016

`http://isabelle.in.tum.de`

**Next release:** December 2016

`http://isabelle.in.tum.de/website-Isabelle2016-1-RC2`